Lesson 10: Packaging, Assemblies, and Namespaces

Michael Murray
Program Manager
Longhorn SDK Team

June 2004

Internal Technical Education

Packaging, Assemblies, and Namespaces

- After successfully completing this lesson, you will be able to:
  - Correctly factor functionality into assemblies
  - Understand assembly best practices (and Microsoft-required assembly conventions)
  - Understand where to install redist and developer bits
  - Understand how to choose good namespaces for your types

Assemblies and Modules

- Modules are physical files (Intermediate Language [IL] and metadata) on disk that must be grouped into assemblies to be used
- Assembly is self-describing "smallest unit of reuse, security, and versioning"
- Typically one assembly contains one module
- Multi-file assemblies are rarely needed in library design—they are good for:
  - Multiple programming languages in the same assembly
  - Streaming downloads

Naming Assemblies

- Use meaningful assembly names, but you don't have to copy the namespace name
  - Simplify "/r" command-line referencing of your library
  - Example: System.Data.Dll
  - Consider using P/L convention for naming:
    - Microsoft.RayTracer.v3.0.dll (library)
    - Microsoft.RayTracer.Dll (App Domain platform)
- Assemblies can contain multiple namespaces and namespaces can span assemblies
  - Optimize each independently

Factoring Assemblies

- Factor functionality into assemblies based on:
  - Performance: There is overhead in loading each assembly—all other things being equal, the fewer assemblies an application loads the quicker the load time
  - Versioning: All code in an assembly must version at the same rate
  - Security: All code in an assembly has the same identity and is granted the same level of trust

Assemblies and Performance

- Prefer single, large assemblies to multiple, smaller assemblies
  - Helps reduce working set of application
  - Large assemblies are easier for NGEN to optimize (better image layout, etc.)
  - If you have several assemblies that are always loaded together, combine into a single assembly
- Tension between versioning/deployment and performance
  - Consider trade-offs: agility vs. raw perf
  - Where is the code used? design time vs. run time

Using NGEN to Start Up Faster

- An NGEN'ed assembly is an assembly that is compiled to native code and saved to disk
- Advantages of NGEN:
  - Single image load (no separate metadata)
  - Class layout already done => faster start up
  - IL already compiled to native—no need to do it again
- Done at install time or on demand
- Reduces start-up time in many cases
- Microsoft® .NET Framework contains several pre-JITed assemblies
  - Mscorlib, Microsoft Windows® Forms, and drawing
- But decrease throughput (7–10%)
  - Native code has version checks and reverts to run-time JIT if they fail
- See NGen.exe in the SDK
- The IL version must be available on client systems

Working Set

- Trim working set
  - For better system performance
  - For better application startup time
- Only load the assemblies you need
  - A single method does not necessarily justify loading an assembly
- Be aware of any assemblies that are loaded as a side effect of you using some API
- Steady state working set will be significantly lower than startup working set

Strong Naming

- Assemblies get their identity from a strong name, which is made up of:
  - Simple Name: The name of the file minus the extension (see "Naming Assemblies" slide)
  - Version: Four-part version number
  - Culture: Used for resource assemblies
  - Public Key and Digital Signature: Establishes the validity of the assembly
- All references to an assembly include all this information (strong binding)

Strong Naming (Continued)

- Always sign your assemblies using a strong name
  - Otherwise, origins cannot be verified
  - CLR cannot warn users if contents of assembly has been altered
  - Can't be loaded into the Global Assembly Cache (GAC)
  - Avoids conflicts between assemblies of the same name and version (if strong name is different)
  - You will need strong name for servicing

Assemblies and Servicing

- **Organizational:** The factoring should enable easy delivery of components from different organizations
- **Ease of Use:** Fewer DLLs are easier to use than many—easier to deliver, set up, build with, and maintain
- **Distribution:** The factoring should help in the creation of SKUs for downloading/distribution
- Do have all code in each assembly version at the same rate

Assembly Version Number

Assembly Version Number:
<Major>.<Minor>.<Build>.<Revision>

- All four parts of the key matter
  - You can't just change the revision and have binding still work
  - Old apps will have to be recompiled or have app policy applied (Caveats: See "Versioning" deck)
- Assemblies should use version numbers—the version number is considered part of an assembly's identity
- Versioning is applied by means of the AssemblyVersion attribute

When to Change Assembly Version Numbers

- Assembly version != File version (necessarily)
  - File version: Changes with each build
  - Assembly version: Usually doesn't change between non-shipping builds
    - Avoids strongly-named assembly loading problems due to version mismatches
    - Can also use publisher policy to bind if version number is different

Assemblies Marked with Common Language Specification (CLS) Compliance

- Assemblies should explicitly state CLS compliance by using the CLSCompliant attribute

```
// Assembly-level CLS Compliant attribute:
[assembly: System.ClsCompliant(true)]
// Class-level CLS Compliant attribute:
[CLSCompliant(true)]
public class MyClass {
    public void Method1() { }
    public void Method2() { }
}
```

For types, may also be used with **false** argument

```
[CLSCompliant(false)]
```

When would you do this?

- Using Generics in your type (not part of CLS yet)
- Exposing methods such as ConvertToUint32(...)

Assembly Info Attributes

- You should always use the info attributes on your assemblies similar to this example:

```
[assembly: AssemblyCompany("Microsoft Corporation")]
[assembly: AssemblyProduct("Microsoft (R) {Your Prod/Tech Name}")]
[assembly: AssemblyCopyright("Copyright (C) Microsoft Corp. 2002")]
[assembly: AssemblyTitle("Friendly name for your assembly")]
[assembly: AssemblyDescription("Brief Description for your
assembly")]
```

Signing Microsoft Assemblies

- Do not ship Microsoft assemblies without having their bits signed by Product Release Services (PRS)—it is against company policy to do so
  - To submit to PRS, go to http://prslab/ and choose "Submit Code"
    - Be sure to select the Microsoft Reusable Component Key under "Strong Name Certificate," then follow the rest of the instructions
    - Typically, this takes an hour for PRS to process, and the worst-case turnaround time is about 4 hours
- Do not use SN.exe to automatically generate the public key
  - Use the contact mentioned above to obtain a Microsoft Reusable Component Key
- Do test the assembly by disabling the CLR signature verification for the assembly
  - This allows build and test systems to install the assembly to the GAC, as well as to bind to the assembly, as though it were signed

Public Key and Digital Signature

- Prevents the assembly from being tampered with after being signed
- Establishes the "identity" of the publisher
  - Allows trust decisions to be made by the administrator
  - Allows publisher policy only from the same publisher

Public Key and Digital Signature

- Must sign with key from Product Release Services (PRS) before public releases
- Temp sign in build lab
- Microsoft has three keys
  - Standards key for libraries standardized through ECMA/ISO
  - System key for libraries needed for base functionality of the runtime
  - MS Shared Library key for libraries published by Microsoft (most teams use this)

IP Protection

- IL/metadata can be dis-assembled into source code
  - But so can x86 code
- They cannot replace your assembly
- If you have IP you must protect
  - Obfuscator
  - Native code
  - Web service
- Really a Digital Rights Management (DRM) issue

When to Use the GAC

- For most libraries: Always install to the GAC
  - GAC = Version-aware Microsoft Win32® directory
  - If you're installing assemblies to be used by multiple apps
  - Easily "patch," one copy per version
- If you're deploying an app or types to be used by just your own app, deploy to the app dir
  - Not impactful on the system

Where to Install

- For redistributable or OS install
  - Install into the GAC
- For design time
  - (1) Install into the GAC
    - F5 runs and design-time controls use these
  - (2) Install SAME bits and XML docs to a versioned subdirectory and add it to the registry
    - For V1.0 assemblies:

      HKLM\Software\Microsoft\VisualStudio\7.0\AssemblyFolder
    - For "Everett" assemblies:

      HKLM\Software\Microsoft\VisualStudio\7.1\AssemblyFolders
  - Compile time uses these as "header file"

Assembly Display Names

- Determines the identity of the assembly at assembling binding time
- Includes the assembly simple name, version, culture, and public key token
- Don't put these together yourself: Allow Fusion to create them for you in canonicalized form, example:

```
System, Version=1.0.3300.0,
  Culture=neutral,
  PublicKeyToken=b77a5c561934e089
```

- Use `Type.AssemblyQualifiedName(…)` etc. to retrieve (or gacutil /l)

Namespaces

- Namespaces are the developer-centered categorization of functionality—INDEX to functionality for developers
- Orthogonal to all the assembly issues
- Identity is based on assembly name, so namespaces don't have to be unique
- Namespaces can span assemblies

Microsoft.* or System.*?

- Use a Microsoft.* namespace if the ship vehicle for your types is a product/SKU outside of the .NET Framework or "Longhorn"/Microsoft WinFX™
- Use a System.* namespace if you are part of "Redist" (Microsoft .NET or "Longhorn")
- See the Design Guidelines document/ "LAPI Namespace Plan" for full guidance

Picking Your Top-Level Namespace

- There are too many top-level namespaces under System already!
- Try to fit within one of the existing "buckets" (examples: System.Windows, System.Net, etc.) if that makes sense (check with owners first)
- Note that namespaces do NOT reflect organization boundaries (if you are on Windows team, that doesn't necessarily mean your types belong in System.Windows)
- Use the API Owner's Tool to "register" your namespace and the owners

Standard Subnamespaces

- Use standard subnamespaces for the following types:
  - .Interop: For native interop types
  - .Advanced: For non-mainline, extensibility, etc. types ("Einstein" scenarios, non-80% case)
  - .Design: Design-time support types
  - .Permission: Security permissions

Exercises: One Assembly or More?

- Given the scenarios that follow, decide if the product should be broken up into separate assemblies and why or why not

Exercises: One Assembly or More?

- Product X has a single library but the classes that make it up are in two different namespaces: Microsoft.Xxx and System.Xxx
- Should these be in separate assemblies? Why or why not?

Exercises: One Assembly or More?

- Product Y has a single library but some of the classes must adhere to a fast-moving industry standard, while the rest of the library will rev much less frequently
- Should these be in separate assemblies? Why or why not?

Exercises: One Assembly or More?

- Product Z shipped one large assembly in V1—many of the classes are not used in a main-line scenario, and they believe that separating these classes into a different assembly will reduce their working set for that scenario
- Should these be in separate assemblies? Why or why not?

Packaging and Namespaces Summary

- Assemblies and namespaces are completely independent
- Always give your assemblies a strong name
- Support side-by-side